

Report

# Assignment -2

---

S. Amna Amir, Tooba Dahar

22i-1142, 22i-1357



## The Approach

### General Flow:

The code reads the cfg from the grammar.txt file, creates a map based on non terminal as key, and productions as value.

Then it calls the leftFactor function on each production.

After leftFactoring, it removes left recursion, and produces the final grammar.

From this grammar, it computes the first sets, and the follow sets.

Using these, it fills in the LL(1) parser table.

### Detailed Explanation (function by function):

#### 1. Input Representation:

The grammar is stored in std::map where the key is a non terminal and the value is the production written as a vector of strings.

#### 2. File Reading:

Splits the line on "-", and "|" to separate different productions.

**Note: The cfg provided should have “->” to indicate start of production, a “|” to indicate multiple productions, and the subsequent productions should always be separated by a space, otherwise they will be read as the same symbol.**

**Lastly, non terminals and terminals are differentiated based on whether they have productions or not.**

#### 3. Tokens:

The split function divides the productions into tokens by splitting whenever there is a space.

#### 4. Finding common prefix:

The common prefix function takes two token's vectors and returns the longest common starting sequence.

This is used for left Factoring

#### 5. Left factoring:

If a common prefix exists, a new non terminal is created, and production for original nonterminal is rewritten.



#### 6. Left recursion:

the productions are split into recursive and non recursive vectors and re-written by using a new non terminal

#### 7. first set:

In a while loop, it iteratively computes the first sets, and continues until no more sets can be changed.

For each non terminal, it adds the terminal found at the beginning of the production.

For non terminals at the beginning, it unions their first sets while excluding epsilon.

#### 8. follow set:

The start symbol's follow is initialized with \$.

then the function iterates over each production, and for every non terminal, it either adds terminals from the first of the following symbols

Or if there are none, then the follow of the left hand non terminal.

#### 9. LL(1) table:

For each production, it computes the first of the non terminal,

and for each terminal  $t$  in the first set, it adds the production to the parsing table

If  $A$  is the starting non terminal, then the production is added in the table in  $table[A][t]$

{ iomanip is used to set widths in the table to make it readable}



## The Challenges

As different data structures such as vectors and maps were required, gaining familiarity with the keywords took time.

Furthermore, ensuring that the code ran on generic CFG required more thinking for splitting of functions and tokenizing.

Creating a follow set based on first and other follow sets was easy in theory but a bit difficult in practice.

Lastly, making the LL(1) table legible required a special library, otherwise, it looked unreadable.

However, the most challenging part of the entire assignment was time management, as juggling assignments and tasks for all subjects, especially with same deadlines, and extensive scopes made otherwise interesting and thought-provoking assignments a burden.

## The Verification

I verified the code by dry running examples and checking if the solutions were equal. Secondly, I used examples from the lecture slides to ensure the outputs matched. Below is an example for demonstration:

### Solution

Find the FIRST and FOLLOW of all the non-terminals:

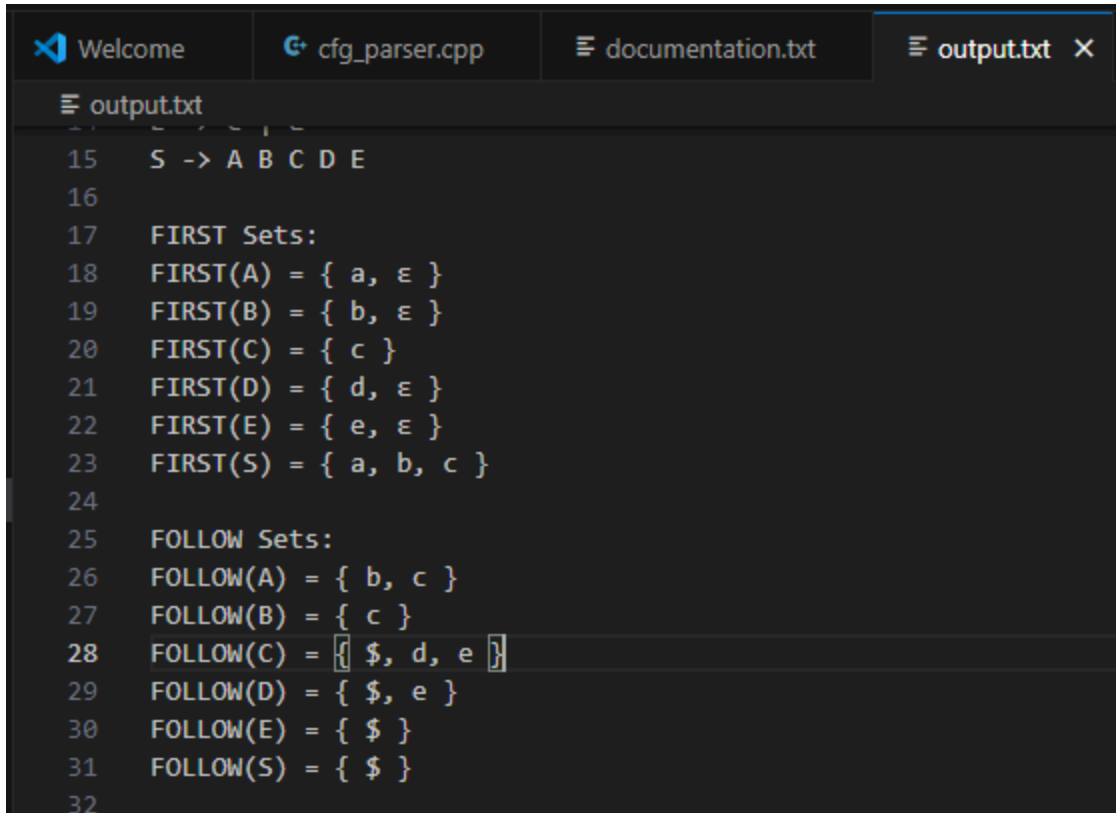
	FIRST	FOLLOW
$S \rightarrow ABCDE$	$\{a, b, c\}$	$\{\$ \}$
$A \rightarrow a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \rightarrow b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \rightarrow c$	$\{c\}$	$\{d, e, \$ \}$
$D \rightarrow d \mid \epsilon$	$\{d, \epsilon\}$	$\{e, \$ \}$
$E \rightarrow e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$ \}$

Above is a picture of an example from the slide.

I pasted in the cfg in the grammar.txt file as shown below:

```
Welcome  cfg_parser.cpp  documentation.txt  output.txt  grammar.txt X
grammar.txt
1  S -> A B C D E
2  A -> a | ε
3  B -> b | ε
4  C -> c
5  D -> d | ε
6  E -> e | ε
```

Next, I ran the code, and matched the output:



```
15  S -> A B C D E
16
17  FIRST Sets:
18  FIRST(A) = { a, ε }
19  FIRST(B) = { b, ε }
20  FIRST(C) = { c }
21  FIRST(D) = { d, ε }
22  FIRST(E) = { e, ε }
23  FIRST(S) = { a, b, c }
24
25  FOLLOW Sets:
26  FOLLOW(A) = { b, c }
27  FOLLOW(B) = { c }
28  FOLLOW(C) = [ $, d, e ]
29  FOLLOW(D) = { $, e }
30  FOLLOW(E) = { $ }
31  FOLLOW(S) = { $ }
32
```